HPC Master Trainer Workshop – February 2024

Introduction to MPI

Sandeep Agrawal

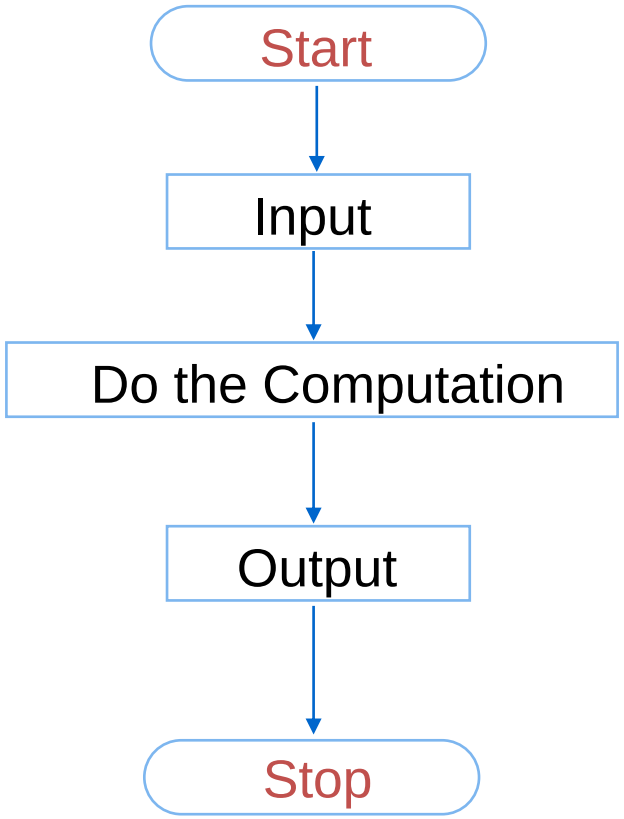
sandeepa@cdac.in

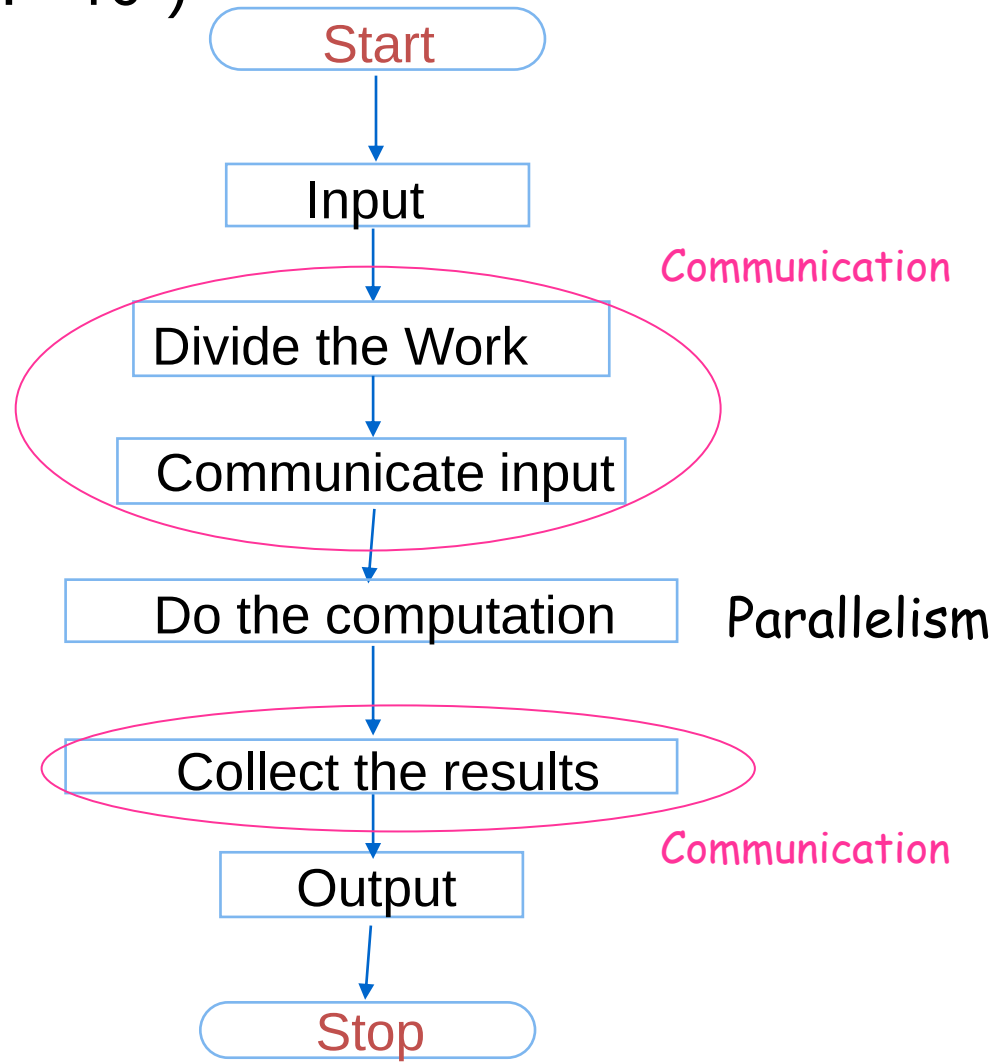
Contents

- General concepts
- About Message Passing Model
- What is MPI
- MPI Point-to-Point Communication
- MPI Datatypes

Serial vs Parallel Flow

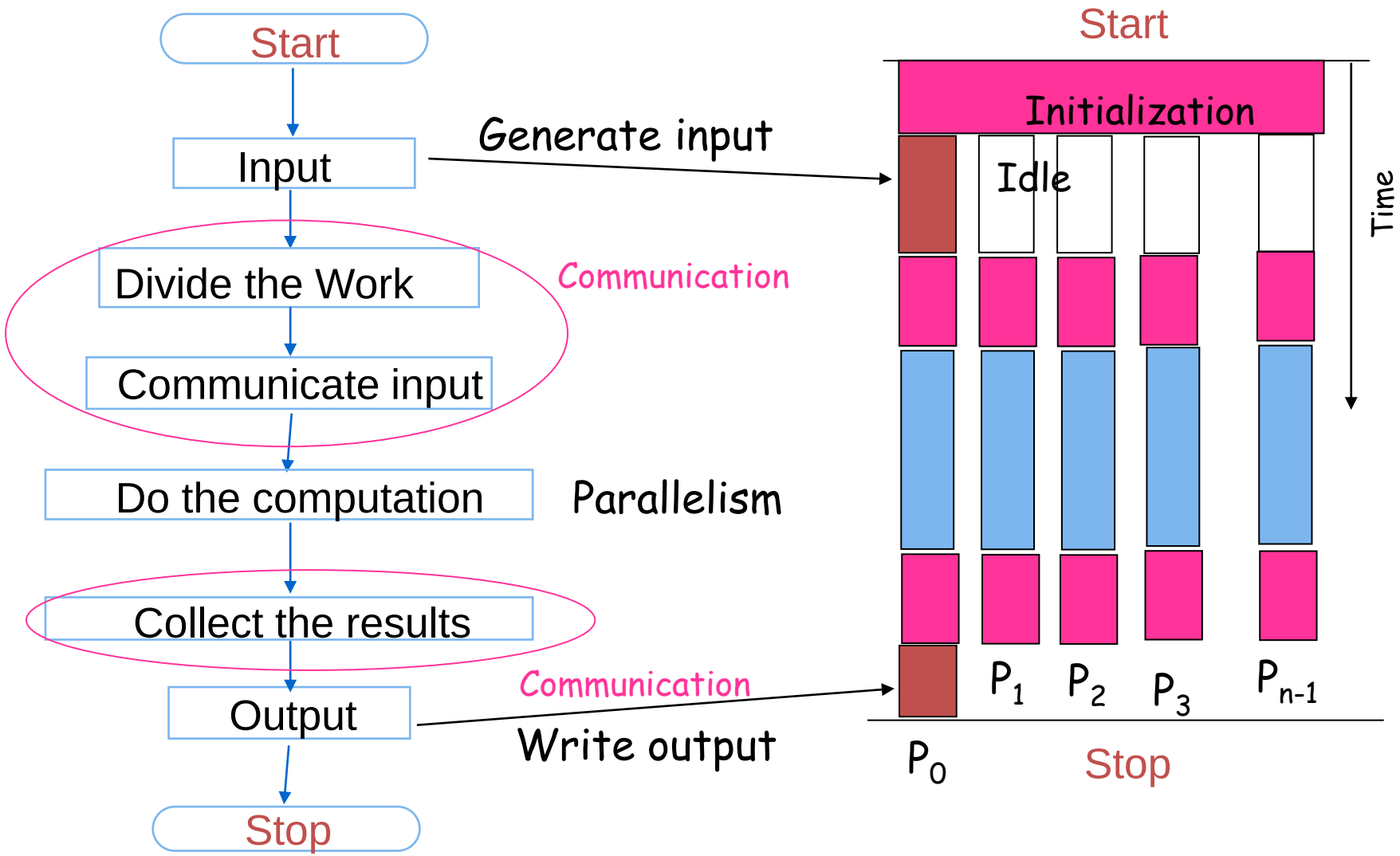
Compute $\sum x_i$ for (i=1,2,…… 10^9)





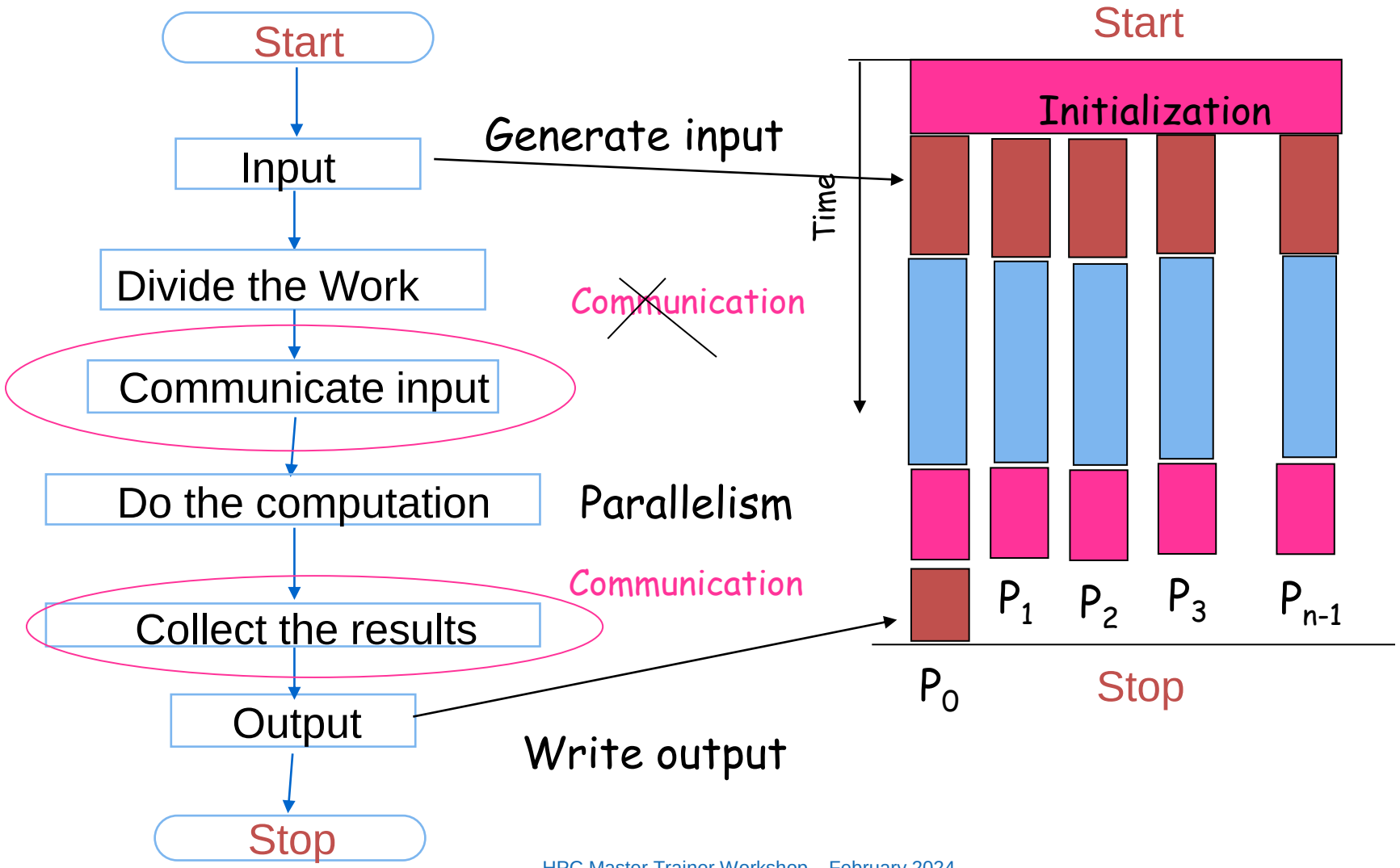
Serial vs Parallel Flow

Option 1: Read and Distribute inputs



Serial vs Parallel Flow

Option 2: Generate inputs simultaneously



Performance Metrics

1. Speed up $= \dfrac{\text{Time for sequential code}}{\text{Time for parallel code}}$

$$S_P = \frac{Ts}{Tp} \qquad 1 \leq S_P \leq P$$

2. Efficiency $E_P = \dfrac{Sp}{p}$ $\qquad 0 < E_P < 1$

$E_P = 1 \Rightarrow S_P = P$ $\qquad$ 100% efficient

Amdahl's Law

$$S = \frac{1}{f + (1-f) / P}$$

f = Sequential part of the code

Example $f = 0.1$ assume $P = 10$ processes

$$S = \frac{1}{0.1 + (0.9) / 10}$$

$$= \frac{1}{0.1 + (0.09)} \cong 5$$

As $P \longrightarrow \infty$ $S \longrightarrow 10$

Whatever we do, 10 is the maximum speedup possible

Communication Overheads

Latency

Startup time for each message transaction e.g. 1 μs

Bandwidth

The rate at which the messages are transmitted across the nodes / processors
e.g. 100 Gbits / sec.

Process and Thread

- An executing instance of a program is called a process
- Process has its independent memory space
- A thread is a subset of the process – also called lightweight process allowing faster context switching
- Threads share memory space within process's memory
- Threads may have some (usually small) private data
- A thread is an independent instruction stream, thus allowing concurrent operation
- In OpenMP one usually wants no more than one thread per core

Characteristics of Message Passing Model

- Separate memory address spaces
- Explicit data and work allocation by user
- Asynchronous parallelism
- Explicit interaction

How Message Passing Model Works

- A parallel computation consists of a number of processes
- Each process has purely local variables
- No mechanism for any process to directly access memory of another
- Sharing of data among processes is done by explicitly message passing
- Data transfer requires cooperative operations by each process
- Different processes need not be running on different processors

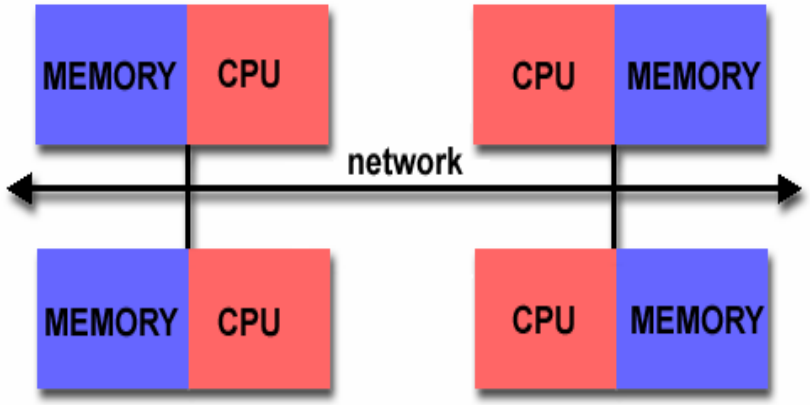
Usefulness of Message Passing Model

- Extremely general model
- Essentially, any type of parallel computation can be cast in the message passing form
- Can be implemented on wide variety of platforms, from networks of workstations to even single processor machines
- Generally allows more control over data location and flow within a parallel application than in, for example the shared memory model
- High performance

What is MPI ?

MPI stands for Message-Passing Interface

- MPI (Message-Passing Interface) is a message-passing library interface specification
- MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process.
- Extensions to the classical message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, threads and parallel I/O
- Every major HPC vendor have their own implementation of MPI
- However, programs written in message-passing style can run that supports such model
    - Distributed or shared-memory multi-processors
    - Networks of workstations
    - Single processor systems

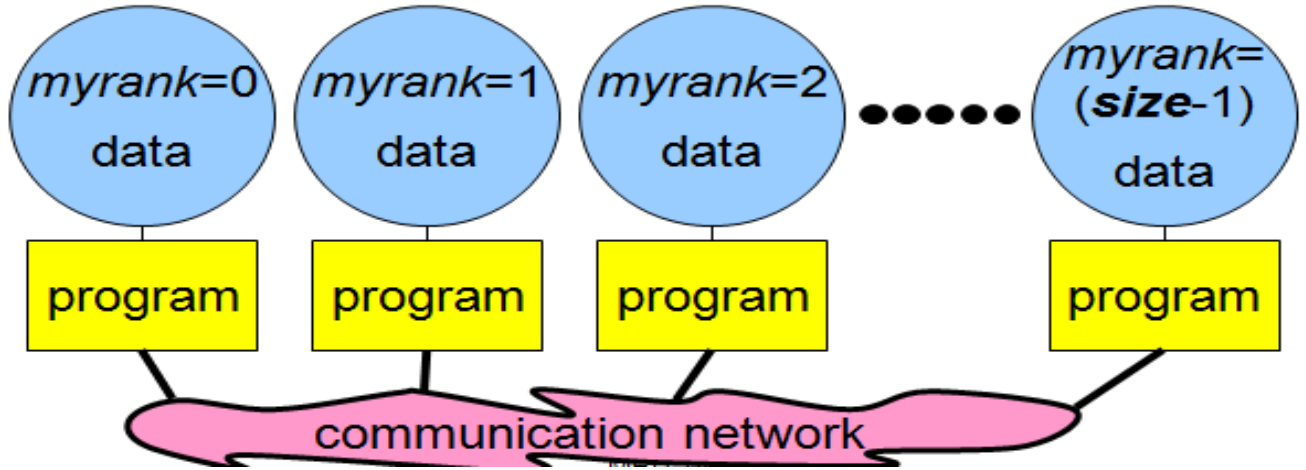


is MPI large or small ?

- MPI is Large (hundreds of functions)
  - Many features require extensive API
  - Complexity of use not related to number of functions
- MPI is small (6 basic functions)
  - All that's needed to get started are only 6 functions
- MPI is just right !
  - Flexibility available when required
  - Can start with small subset

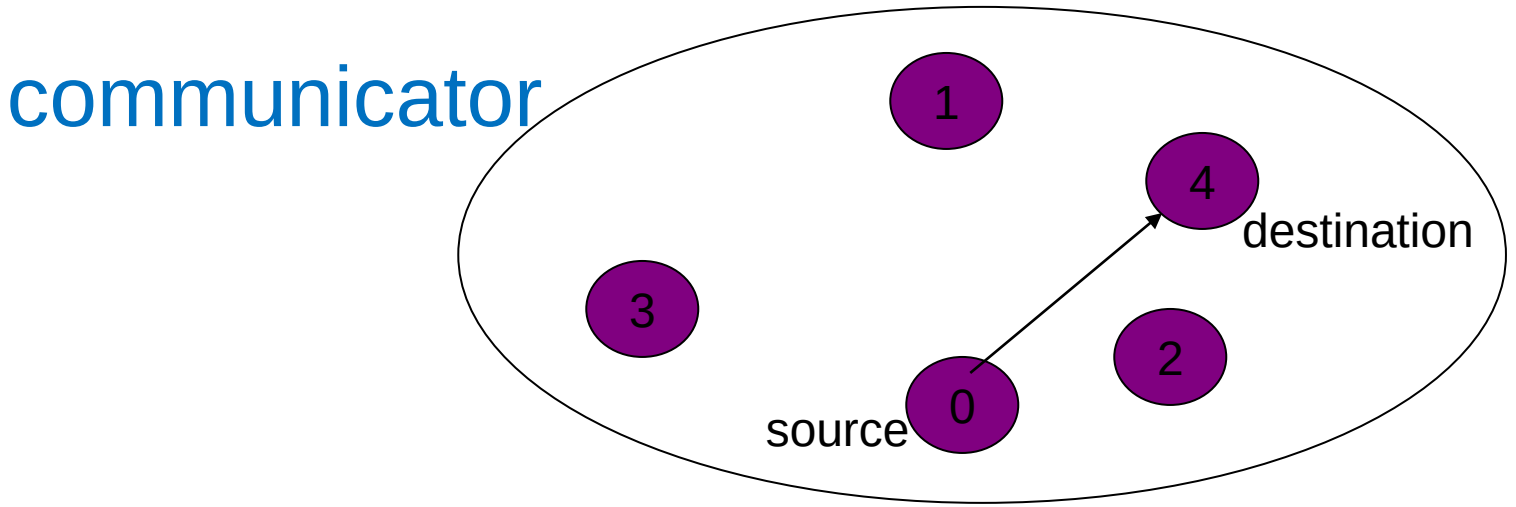
Data and Work Distribution

- Programmer imagines several processors, each with own memory, and writes a program to run on each processor
- To communicate together mpi-processes need unique identifiers: rank = identifying number
- all distribution decisions are based on the *rank*
  - like which process works on what data
  - which process works on what tasks



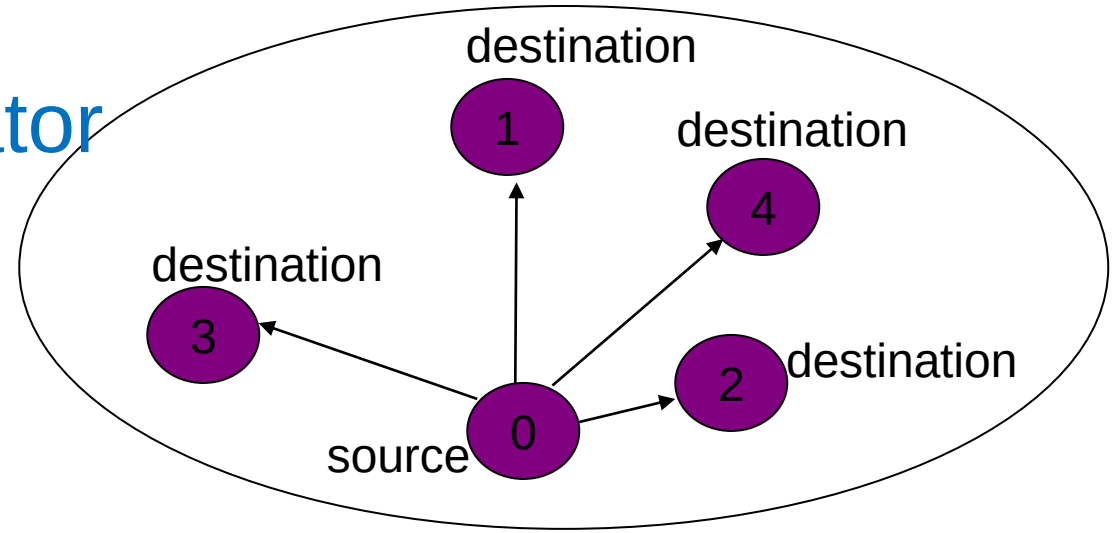
Point-to-Point Communication

- Communication between two processes
- Source process sends message to destination process
- Communication takes place within a communicator
- MPI_COMM_WORLD is default communicator



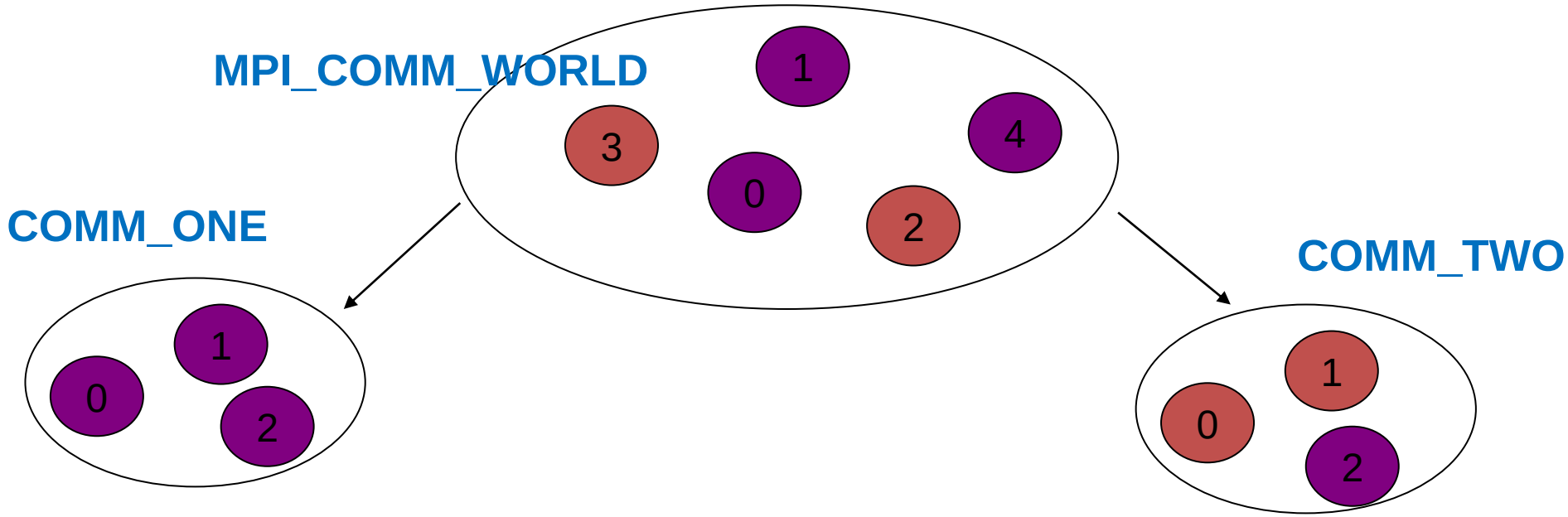
Collective Communication

- Communication between all processes
- A source process sends messages to or receives messages from all other processes
- Communication takes place within a communicator
- MPI_COMM_WORLD is default communicator



Communicators

- Is an object to handle a collection of processes
- Only processes within a communicator can talk among themselves
  - **ranks 0 to N-1**
- MPI_COMM_WORLD is default communicator containing all the processes
- User can create subsets of default communicators – overlapping or disjoint



Building blocks: Send and Recv

Basic operations in Message-passing programming paradigm are send and receive

```
send(void *sendbuf,  int noelems,  int dest)
```

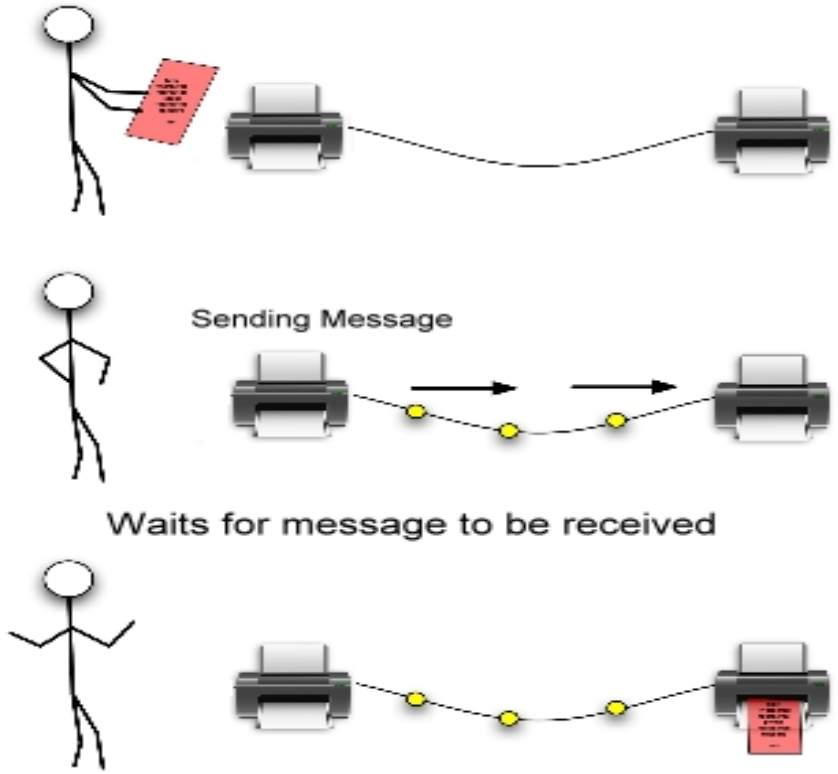
```
receive(void *recvbuf,  int noelems,  int source)
```

Building blocks: Send and Recv (contd….)

- “Completion” means that memory locations used in the message transfer can be safely accessed
    - send: variable sent can be modified after completion
    - receive: variable received can now be used
- MPI communication modes differ in what conditions on the receiving end are needed for completion
- Communication modes can be blocking or non-blocking
    - Blocking: return from function call implies completion
    - Non-blocking: routine returns immediately, completion to be tested for

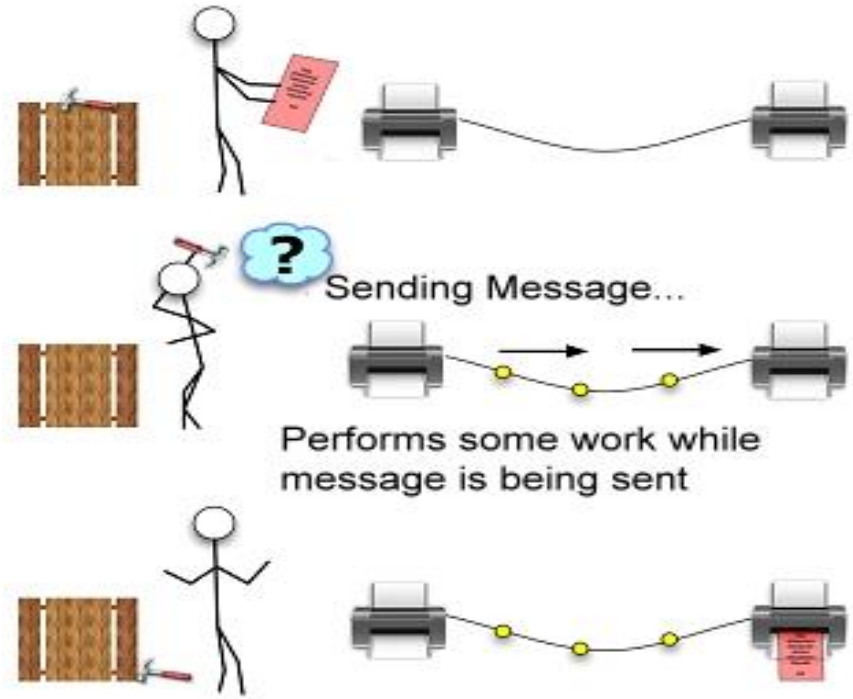
Blocking Operation

An operation that does not complete until the operation either succeeds or fails



Non-Blocking operation

An operation, such as sending or receiving a message, that returns immediately whether or not the operation was completed



Point to Point Communication

- Message is sent from a sending process to a receiving process. Only these two process need to know anything about the message.

- Message passing system provides following information to specify the message transfer
    - Which process is sending the message
    - Where is the data on the sending process
    - What kind of data is being sent
    - How much data is there
    - Which process is receiving the message
    - Where should the data be left on the receiving process
    - How much data is receiving process prepared to accept

General MPI Program Structure

MPI include file

.
.
.

Initialize MPI environment

.
.
.

Do work and make message passing calls

.
.
.

Terminate MPI Environment

Header files and calls format

- MPI constants, macros, definitions, function prototypes and handles are defined in a header file
- Required for all programs/routines which make MPI library calls

C (case sensitive):

```
# include “mpi.h”

error = MPI_Xxxxx(parameter,...);
```

Fortran (case unimportant):

```
include “mpif.h”

CALL MPI_XXXXX(parameter,...,IERROR)
```

Starting With MPI Programming

- Six basic functions to start :

1. MPI_INIT	Initialize MPI Environment
2. MPI_FINALIZE	Finish MPI Environment
3. MPI_COMM_RANK	Get the process rank
4. MPI_COMM_SIZE	Get the number of processes
5. MPI_Send	Send data to another process
6. MPI_Recv	Get data from another process

Initializing MPI

- MPI_Init is the first MPI routine called (only once)
- Initializes the MPI environment

C: int MPI_Init(int *argc, char ***argv)

Communicator Size

- How many processes are contained within a communicator?

C: MPI_Comm_size (MPI_Comm comm, int *size)

Process Rank

- Process ID number within the communicator
  - Starts with zero and goes to (n - 1) where n is the number of processes requested
- Used to identify the source and destination of messages

C: MPI_Comm_rank(MPI_Comm comm, int *rank)

Exiting MPI

- Performs various clean-ups tasks to terminate the MPI environment.
- Always called at end of the computation.

C: MPI_Finalize()

Note : If any one process does not reach the finalization statement, the program will appear to hang.

Example program: hello_world.c

```
#include "mpi.h"
#include <stdio.h>
int main( argc, argv)
int argc; char **argv;
{
int rank, size;
MPI_Init( &argc, &argv );
MPI_Comm_rank(MPI_COMM_WORLD, &rank );
MPI_Comm_size(MPI_COMM_WORLD, &size );
/* Your code here */
printf("Hello world! I'm %d of %d\n", rank, size);
MPI_Finalize();
return 0;
}
```

Header File

Communicator

Initializing MPI

Rank

Size

Exiting MPI

Example program 1: hello_world.f

```
program hello
include 'mpif.h'

integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

print*, 'node', rank, ': Hello world '

call MPI_FINALIZE(ierror)

end
```

How to Compile &Execute MPI Programs ?

To Compile : mpicc hello_world.c -o hello

mpif90 hello_world.f -o hello

To run with 4 processes : mpiexec -np 4 hello

Output Hello world! I'm 2 of 4

Hello world! I'm 1 of 4

Hello world! I'm 3 of 4

Hello world! I'm 0 of 4

Note - Order of output is not specified by MI

MPI Send

```
int MPI_Send( void *buf,          // Data To be sent
              int count,          // Total Data Elements to be sent
MPI_Datatype datatype,            // Datatype of the data to be sent
              int dest,           // Processor to which data is being sent
              int tag,            // To distinguish from diff types of msg
          MPI_Comm comm)          // Communicator
```

MPI Receive

```
int MPI_Recv(void *buf,     // Data To be Receive
             int count,     // Total Data Elements to be recv
MPI_Datatype datatype,  // Datatype of the data to be recv
             int source,    // Processor from where data is being sent
             int tag,       // To distinguish from diff types of msg
         MPI_Comm comm,     // Communicator
        MPI_Status *status)    // Status structure
```

Wildcards

- Allow you to not necessarily specify a tag or source
  - Eg :MPI_ANY_SOURCE and MPI_ANY_TAG are wild cards
- Status structure is used to get wildcard values
  - The tag of a received message

    C : status.MPI_TAG
  - The source of a received message

    C : status.MPI_SOURCE
  - The error code of the MPI call

    C : status.MPI_ERROR

MPI Status

Accessing status information

- The tag of a received message
  - C : status.MPI_TAG
  - Fortran : STATUS(MPI_TAG)
- The source of a received message
  - C : status.MPI_SOURCE
  - Fortran : STATUS(MPI_SOURCE)
- The error code of the MPI call
  - C : status.MPI_ERROR
  - Fortran : STATUS(MPI_ERROR)

Message Datatype

- A message contains an array of elements or scalar element of some particular MPI datatype
- MPI datatypes:
  - Basic types
  - Derived types
- Derived types can be build up from basic types

MPI DataTypes

C Data Types		Fortran Data Types	
MPI_CHAR	signed char	MPI_CHARACTER	character(1)
MPI_SHORT	signed short int		
MPI_INT	signed int	MPI_INTEGER	integer
MPI_LONG	signed long int		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED_SHORT	unsigned short int		
MPI_UNSIGNED	unsigned int		
MPI_UNSIGNED_LONG	unsigned long int		
MPI_FLOAT	float	MPI_REAL	real
MPI_DOUBLE	double	MPI_DOUBLE_PRECISION	double precision
MPI_LONG_DOUBLE	long double		
		MPI_COMPLEX	complex
		MPI_DOUBLE_COMPLEX	double complex
		MPI_LOGICAL	logical
MPI_BYTE	8 binary digits	MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack	MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

Sending a Message

- int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - buf: starting address of the data to be sent
  - count: number of elements to be sent (not bytes)
  - datatype: MPI datatype of each element
  - dest: rank of destination process
  - tag: message identifier (set by user)
  - comm: MPI communicator of processors involved
- MPI_Send(data, 500, MPI_FLOAT, 5, 25, MPI_COMM_WORLD)

Receiving a Message

- int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  - buf: starting address of buffer where the data is to be stored
  - count: number of elements to be received (not bytes)
  - datatype: MPI datatype of each element
  - source: rank of source process
  - tag: message identifier (set by user)
  - comm: MPI communicator of processors involved
  - status: structure of information about the message that is returned

- MPI_Recv(buffer, 500, MPI_FLOAT, 3, 25, MPI_COMM_WORLD, status)

Blocking Communication Functions

Mode	MPI Function
Standard send	MPI_Send
Synchronous send	MPI_Ssend
Buffered send	MPI_Bsend
Ready send	MPI_Rsend
Receive	MPI_Recv

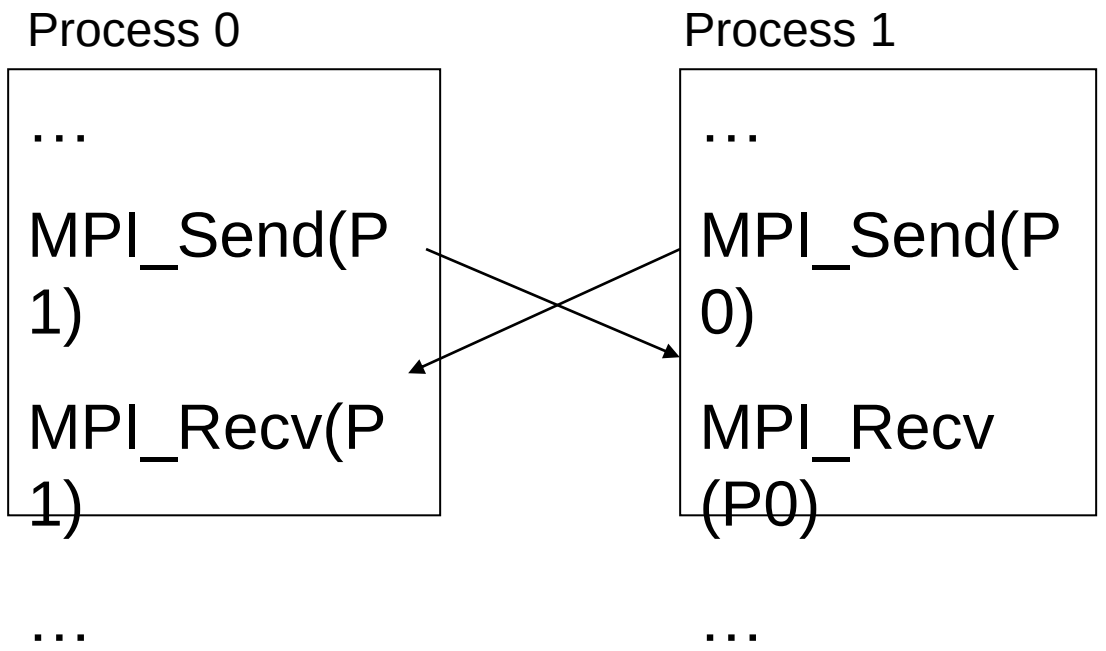
Similar variants exist for non-blocking calls also

For a Communication to Succeed

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Receiver’s buffer must be large enough
- User-specified buffer should be large enough (buffered send only)
- Receive posted before send (ready send only)

Deadlocks

- A deadlock occurs when two or more processors try to access the same set of resources
- Deadlocks are possible in blocking communication
  - Example: Two processors initiate a blocking send to each other without posting a receive



MPI Programming Style

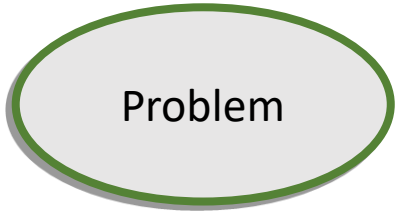
SPMD : Single program multiple data

Each process executes the same program but with different input data

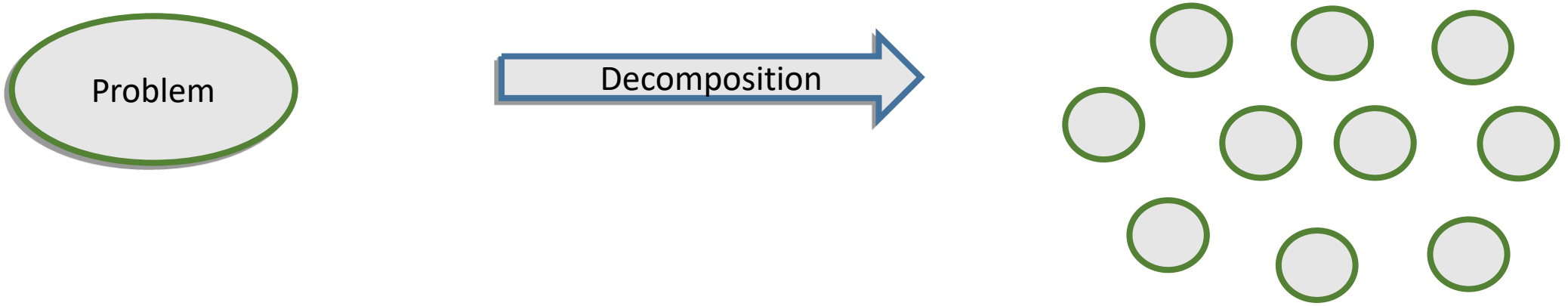
MPMD : Multiple programs multiple data

Different processes may execute different programs

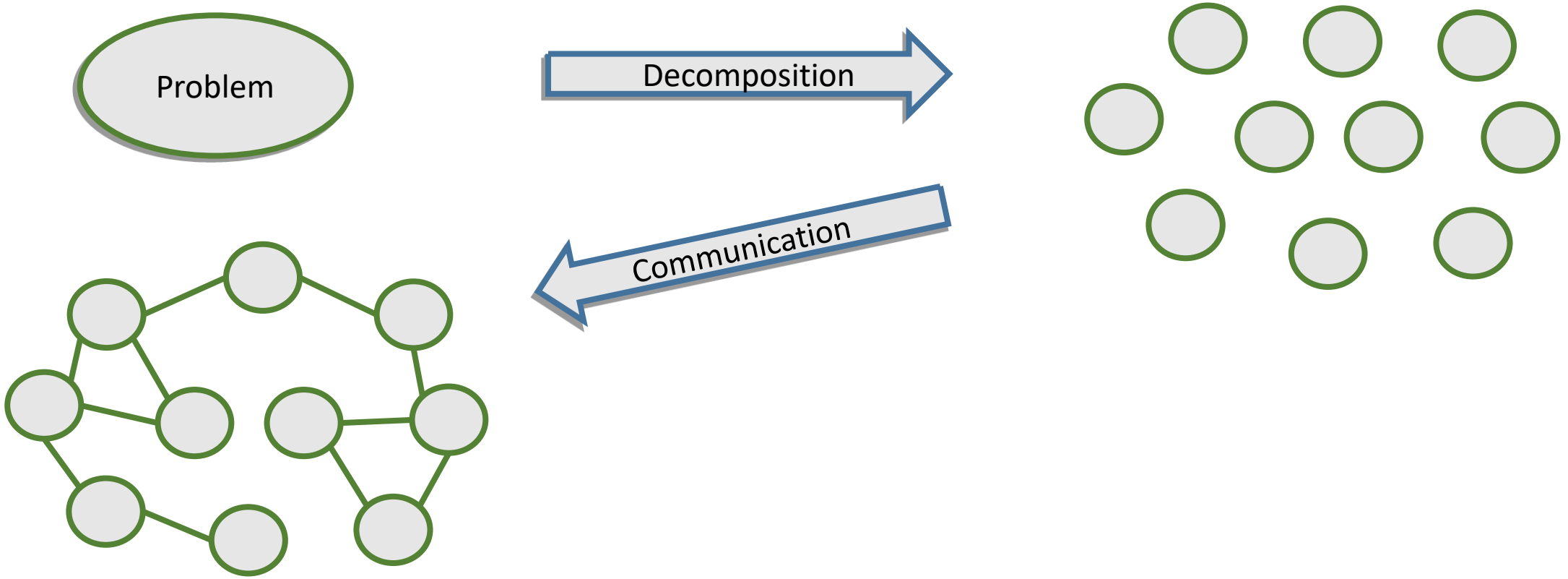
Methodology to Develop Parallel Applications



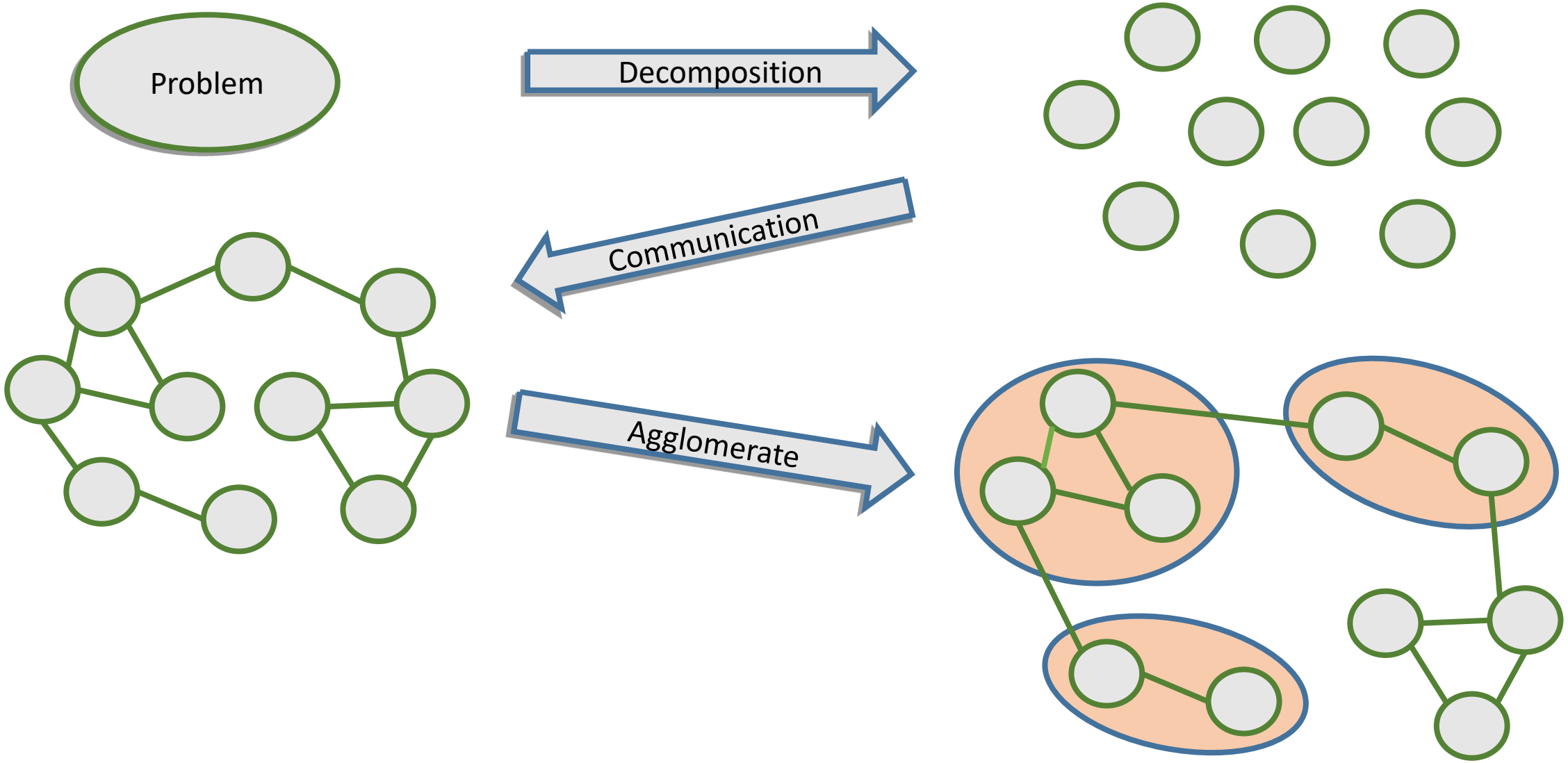
Methodology to Develop Parallel Applications



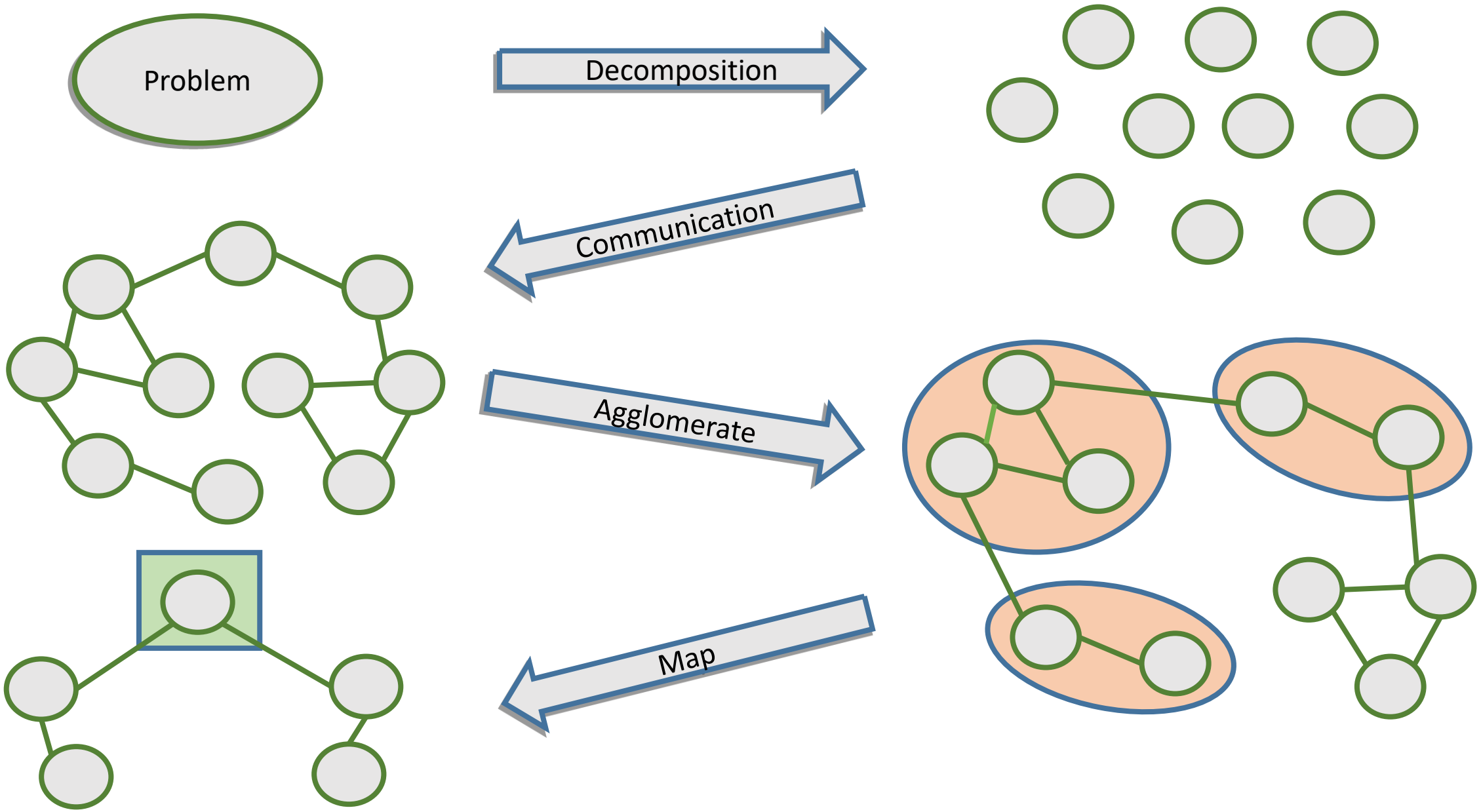
Methodology to Develop Parallel Applications



Methodology to Develop Parallel Applications

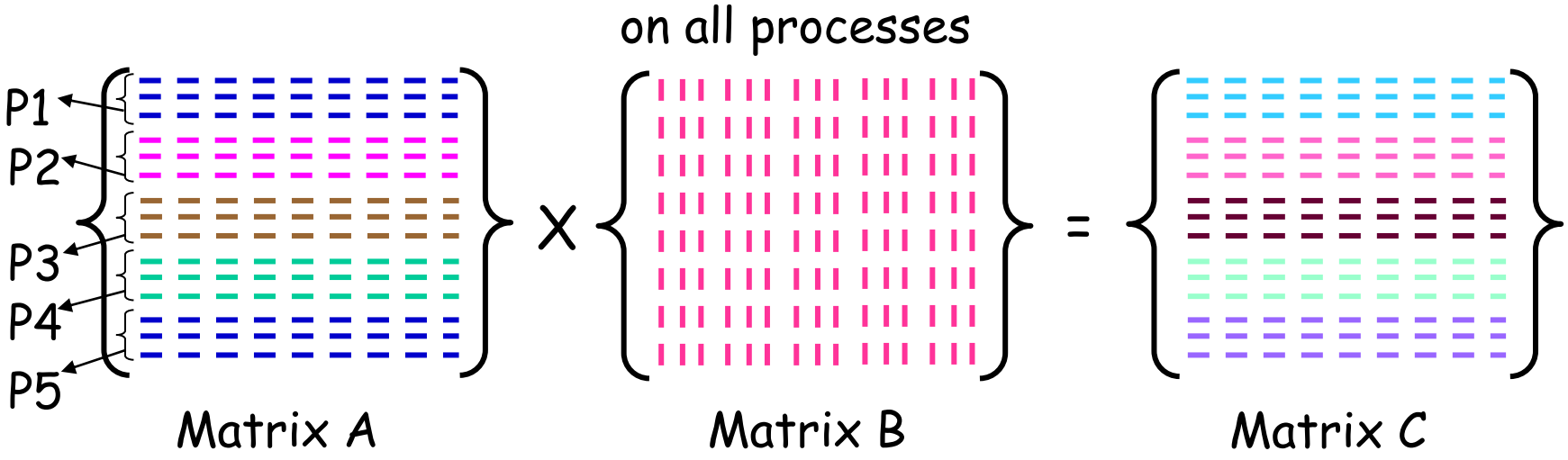


Methodology to Develop Parallel Applications



Defensive Matrix Multiplication

- The master process is the controller process that
  - Distributes Matrix A (row-wise) to each worker process
  - Communicates complete Matrix B to each worker process
- Each worker multiplies assigned rows of matrix A with matrix B
- Master process collects resultant Matrix C from each worker process.



Example program: Matrix-Matrix Multiply

```
#define NRA 50 /* Rows in Matrix A */
#define NCA 40 /* Columns in Matrix A */
#define NCB 30 /* Columns in Matrix B */
#include "mpi.h"
#include <stdio.h>

int main(int argc, int *argv[])
{
  int numtasks,taskid; /* No.of tasks and task identifier */
  int source,dest      /* Task id of message source and destination*/
  double a[NRA][NCA],b[NCA][NCB],c[NRA][NCB]; /* Matrix A, B, C */
  rows,averow,extra,offset,numworkers,i,j,k,rc; /* Miscellaneous */
  MPI_Status status;

  rc = MPI_Init(&argc,&argv);
  rc|= MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  rc|= MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
  if (rc != 0)printf("\nError initializing MPI or Task ID\n");
  else printf("\nTask ID = %d\n", taskid);
  numworkers = numtasks-1;
```

Master process

```
if(taskid == 0)
{
  printf("Number of worker tasks = %d\n",numworkers);
  for (i=0; i<NRA; i++)   /* Generate data for Matrix A & B */
    for (j=0; j<NCA; j++)a[i][j]= i+j;
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)b[i][j]= i*j;

  averow = NRA/numworkers;
  extra = NRA%numworkers;
  offset = 0;
  for (dest=1; dest<=numworkers; dest++)
  {
    rows = (dest <= extra) ? averow+1 : averow;
    printf("\nSending %d rows to task %d\n",rows,dest);
    MPI_Send(&offset,      1,       MPI_INT,   dest,1,MPI_COMM_WORLD);
    MPI_Send(&rows,        1,       MPI_INT,   dest,1,MPI_COMM_WORLD);
    MPI_Send(&a[offset][0],rows*NCA,MPI_DOUBLE,dest,1,MPI_COMM_WORLD);
    MPI_Send(&b,           NCA*NCB, MPI_DOUBLE,dest,1,MPI_COMM_WORLD);
    offset = offset + rows;
   }
```

Master process (contd…)

```
  for (i=1; i<=numworkers; i++) /* Wait for results from workers */
  {
    source = i;
    MPI_Recv(&offset,1,MPI_INT,source,2,MPI_COMM_WORLD,&status);
    MPI_Recv(&rows,  1,MPI_INT,source,2,MPI_COMM_WORLD,&status);
    MPI_Recv(&c[offset][0],rows*NCB,MPI_DOUBLE,source,2,
             MPI_COMM_WORLD, &status);
  }

  printf("Here is the result matrix\n"); /* Print Results */
  for (i=0; i<NRA; i++)
  {
    printf("\n");
    for (j=0; j<NCB; j++)
      printf("%6.2f ", c[i][j]);
  }
  printf ("\n");
}
```

Worker processes

```
if (taskid > 0) /* Worker Tasks */
{
  MPI_Recv(&offset,1,       MPI_INT,    0,1,MPI_COMM_WORLD,&status);
  MPI_Recv(&rows,  1,       MPI_INT,    0,1,MPI_COMM_WORLD,&status);
  MPI_Recv(&a,     rows*NCA,MPI_DOUBLE, 0,1,MPI_COMM_WORLD,&status);
  MPI_Recv(&b,     NCA*NCB, MPI_DOUBLE, 0,1,MPI_COMM_WORLD,&status);
  for (k=0; k<NCB; k++)
    for (i=0; i<rows; i++)
    {
      c[i][k] = 0.0;
      for (j=0; j<NCA; j++)
        c[i][k] = c[i][k] + a[i][j] * b[j][k];
    }
  MPI_Send(&offset, 1,        MPI_INT,    0, 2, MPI_COMM_WORLD);
  MPI_Send(&rows,   1,        MPI_INT,    0, 2, MPI_COMM_WORLD);
  MPI_Send(&c,      rows*NCB, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
}
 MPI_Finalize();
} /* End main */
```


प्रशंसावादः
धन्यवाद
આભાર
Thank You
धन्यवाद धन्यवाद धन्यवाद